Formalizing Routing Models in ACL2

Warren A. Hunt, Jr. Robert Bellarmine Krug Matt Kaufmann Sandip Ray

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712, USA
{hunt,rkrug,kaufmann,sandip}@cs.utexas.edu

Abstract

We present two preliminary formalizations of router networks, both expressed in the logic of the ACL2 theorem prover. One formalization focuses on connectivity requirements by formalizing *validity*, *visibility*, and a trivial example routing policy, and demonstrates the ability to execute specifications. The other formalization focuses on network security properties, specifically *information-flow* and *non-interference*, and includes theorems demonstrating these properties for a simple formalization of a router network.

1 Introduction

We present two formalizations for router networks, each performed using the ACL2 theorem prover [3, 4]. Each formal model is self-contained, independent of the other. Section 2 illustrates how one can use ACL2 to formalize connectivity notions, in particular *validity* and *visibility* [1], and a trivial, related routing policy. Moreover, we demonstrate how to check such notions by fast execution on specific network configurations. The second effort, in Section 3, presents proofs of security properties: specifically, an *information flow* theorem showing that *marked* information only flows to *marked* nodes, and a *noninterference* corollary that says that a low-security viewer cannot see the propagation of high-security information under the assumption that high-security information is never delivered to a low-security router node.

Detailed documentation and explanation may be found as comments in the corresponding two ACL2 *books*: files successfully *certified* by ACL2. See the appendices. As such the body of the paper itself contains minimal explanation of our models and proofs, focusing only on the motivational aspects.

Although our two exercises are independent, they share a common basic framework as we now describe. We consider a network to consist of routers. One could also consider *hosts*, where a host gives a router a message packet and asks it to be routed to some destination. But in our models we draw no explicit distinction between hosts and routers — a host is simply connected to a single neighbor, presumably a router, while a router is typically connected to more than one neighbor. (A host would therefore have a very minimal routing table.)

Modern routers perform much more than just routing packets to intended destinations. They exchange routing information with other routers, provide for IP name translation and port mapping, enforce routing policies, control bandwidth allocations, filter packets based on source-destination information and content, perform encryption/decryption, and enforce port access restrictions. We ignore these aspects in our exercises; while our simple models are not realistic abstractions of the Internet, they do provide starting points for our investigations.

- A *system state* contains a collection of *nodes*. Each node has a router function. Each node can potentially serve as a host by maintaining a local state.
- Node state fields include local states and message queues.

For each formalization we define an *interpreter* of the following form, where `step` is a *transition function* mapping a state and current input to a *next state*.

$$\mathit{run}(s,\mathcal{E}) \triangleq \begin{cases} s & \mathit{if}\ \mathcal{E}=[] \\ \mathit{run}(\mathit{step}(s,\mathit{first}(\mathcal{E})),\mathit{rest}(\mathcal{E})) & \text{otherwise} \end{cases}$$

In the remainder of the paper as well as the books in the appendices, we often refer to the sequence of current inputs an an *oracle*.

Our first model accounts for nodes joining or leaving the network, while the second does not. A second difference is that when creating a new packet, the first model constructs the packet solely by consulting the local state of the router node that is creating the packet, while the second model consults the current oracle input for the destination of the packet. This second difference is intentional: the first model is perhaps more realistic in this regard, but the second model simplifies the statements of our information-flow and noninterference theorems, as follows. Theorems about the second model are formalized by considering notions of system state equivalence based on "erasing" information that is supposed to be irrelevant, such as the contents of high-security packets when we are considering system views available to low-security viewers. If packet destinations can depend on system state, then equivalent initial states can transition by generating a high-security packet headed to two different nodes in the two systems. We would thus need to ignore not only packet contents in the message queues, but entire high-security packets. Such formalization and proof seems feasible but is beyond the scope of what we wanted to try initially.

This work constitutes small first steps towards our ultimate goal, which involves building *stacks* of successively more concrete network routing models M_0, M_1, M_2, ... This paper presents two models with associated properties, such that could each be viewed as M_0, the top of a stack. Lower layers of a stack are to refine the higher layers. Thus lower layers are to formalize routing protocols such as BGP, IGP, and OSPF, while the lowest layer is to formalize network protocols by formally defining the TCP and IP specifications. The goal is to prove that the lowest layer operates in manner that satisfies the abstract specifications at the highest layer. Thus our intention is to develop a number of stacks, all with the same lower layers but with different higher layers depending on the abstract properties to be studied, for example connectivity and security in the respective two models presented in this paper.

2 Connectivity

The first ACL2 book takes us through a very simplified model of a network of routers, together with formalizations of *validity* and *visibility* and a trivial example routing policy. It also illustrates how one can execute such predicates on a formalized network of routers.

Informally, a router network satisfies *validity* if each routing table path is a physical path in the network, i.e., if each hop specified by a routing table is an edge in the actual network graph. *Visibility* is a sort of converse notion, stating that for every pair of nodes A and B in the network, if there is some physical network path from A to B, then packet sourced at A with destination B will actually arrive at B when following the path specified by the routing tables.

We include an example routing policy merely to give a sense of how these can be formalized in ACL2. This policy, which is unrealistically simple, is based on a formalization of *autonomous system*, or *AS*, which is just a set of nodes, and where the network is partitioned into disjoint ASes. Our example policy states that every path of length n from a router in autonomous system $AS(i)$ to a router in $AS(k)$ goes through $AS(j)$.

We defer further explanation to Appendix A. There, one can find our ACL2 formalizations of validity (function `system-valid`) and visibility (function `system-visibility`). The book concludes with an ACL2 formalization of the above routing policy.

A tricky aspect of the book is *executable encapsulates* by way of a macro, `encap`. These allow for predicates to be constrained rather than defined, and yet they are in some sense executable by use of the witness functions and Lisp *features*. The default case does not have the feature `acl2-exec` set in the environment, resulting in `encap` behaving as an ordinary ACL2 `encapsulate`, i.e., to introduce constrained functions with specified properties. If instead the feature `acl2-exec` is set, then `encap` instead *defines* functions that satisfy the specified constraints. Thus, with the feature off one can do proofs in the normal manner, but one can instead turn the feature on and do some testing.

3 Security Properties

The goal of this exercise is to experiment with how to formalize and prove security policies for routers (and firewalls and similar artifacts). Here we formalize a very simple static router configuration in ACL2 and prove a simple information flow property, from which we derive a simple non-interference security property.

Imagine now that we divide the nodes into *low security* (or *low*) nodes and *high security* (or *high*) nodes. Imagine that we enforce the following *write-up* policy: a packet created by a high node must never have a low node as its destination. Then we want to prove the following theorem of the classic *non-interference* style [5, 2], which informally states that a low-security observer (node) cannot detect high-security information. For this theorem, we call two states *low-equivalent* if they agree at every node except perhaps for the local state of each high node and the content of each packet with a high source. An initial state is one whose message queues are empty.

Theorem 1. (Non-interference) *Suppose we run our interpreter on low-equivalent initial states st1 and st2 with the same input sequence (oracle), where no element of that sequence creates a packet from a high source to a low destination. Then the two resulting states are low-equivalent.*

More generally, imagine that some nodes are *marked*, meaning that their local states should be ignored. For example, the "low" vs. "high" discussion above suggests that the high nodes are marked, since they should be ignored from a low perspective. More generally, we may think of marked nodes as nodes to which certain information has propagated. Also, imagine that packets are marked if they are created from marked nodes; and, any node that accepts a marked packet is henceforth marked. Finally, let's say that two states are *marked-equivalent* if they agree when we ignore the local states of all marked nodes and ignore the contents of all marked packets.

Theorem 2. (Information flow) *Suppose we run our interpreter on marked-equivalent initial states st1 and st2 with the same inputs (oracle). Then the two resulting states are marked-equivalent.*

We actually prove slightly stronger theorems by eliminating the restrictions to initial states, using suitable inductive invariants that comprehend the queues. We then obtain the stronger non-interference theorem as a corollary of the stronger information flow theorem, by viewing a node (respectively, packet) as marked if it is a high node (respectively, has a high source). For if no packet is created to go from a high to a low node, then the set of marked nodes remains constant, and hence since the two final states are marked-equivalent (by the latter theorem), they are low-equivalent with respect to the original set of high nodes. Finally, the theorems above about initial states are obtained as corollaries.

4 Conclusion and Future Work

We have only touched the surface of the problem of specifying routing networks and verifying their properties. Our work here focuses on different possible abstract layers of a stack of network models and specifications. We have demonstrated how to use ACL2 to create such models and properties, to verify these properties with its mechanical prover, and to use its execution capability for effective simulation. Future work will focus both on a wider and more realistic range of abstract models and specifications, as well as development of more concrete models and suitable correspondence theorems. It will be interesting to see how to improve the reasoning infrastructure to manage these larger efforts, including ACL2 enhancements, use of external tools, and proof strategy tools.

Acknowledgements

This material is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591. We thank Mike Dahlin, Mohamed Gouda, and Vitaly Shmatikov for discussions on effective formalization of routing networks.

References

[1] N. Feamster and H. Balakrishnan. Towards a Logic for Wide-Area Internet Routing. In J. Wroclawski, editor, *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, August 2003.

[2] J. T. Haigh and W. D. Young. Extending the Noninterference Version of MLS for SAT. *IEEE Transactions on Software Engineering*, 13(2):141–150, 1987.

[3] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, June 2000.

[4] M. Kaufmann and J S. Moore. ACL2 home page, 2008. `http://www.cs.utexas.edu/users/moore`.

[5] J. Rushby. Proofs of Separability — A Verification Technique for a Class of Security Kernels. In *Proceedings of the 5th International Symposium on Programming*, volume 137 of *LNCS*, pages 352–367. Springer-Verlag, 1982.

A Book for Connectivity

This book is broken into sections as follows.

> **Section 1**: Executable constrained functions
> **Section 2**: A trivial, generic model of a network
> **Section 3**: Formalization of network properties: Validity and Visibility
> **Section 4**: Sample simulations
> **Section 5**: Defining a routing policy

```
; In order to execute with this file:
; (push :acl2-exec *features*)

(in-package "ACL2")

; NOTE: For faster execution we could consider finishing the guard
; verification.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section 1: Executable constrained functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; A basic macro is encap, a version of encapsulate that creates non-local
; executable functions in a special #+acl2-exec mode.  Defun-loc is just a
; local defun normally, but in #+acl2-exec mode it's a defun.

(defmacro defun-loc (name &rest args)
  #-acl2-exec
  `(local (defun ,name ,@args))
  #+acl2-exec
  `(progn (defun ,name ,@args)

; Turn off everything about these functions, so that proofs are done about the
; same regardless of #+acl2-exec.  (As Jared Davis has pointed out, these
; functions would still be executed when called in raw Lisp by callers defined
; with ordinary defun.)

          (in-theory (disable ,name (,name)))))

(defmacro encap (sigs &rest events)
  #-acl2-exec
  `(encapsulate ,sigs ,@events)
  #+acl2-exec
  (declare (ignore sigs))
  #+acl2-exec
  `(encapsulate () ,@events))

(defmacro defthm-exec (&rest args)

; Use this macro when you want to prove the theorem only when #+acl2-exec, and
; then, you want to prove it by execution alone.
```

```
  #-acl2-exec
  `(value-triple '(skipped (defthm ,@args)))
  #+acl2-exec
  `(defthm ,@args
     :hints (("Goal"
              :in-theory
              (union-theories (executable-counterpart-theory :here)
                              (current-theory :here))))
     :rule-classes nil))

(defmacro defconst-exec (&rest args)
  #-acl2-exec
  `(value-triple '(skipped (defconst ,@args)))
  #+acl2-exec
  `(defconst ,@args))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section 2: A trivial, generic model of a network
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; This section just provides a very basic model of routing.  No policy or
; security or any other thing is modeled at this level.  A router merely gets a
; message and forwards the message according to the destination of the message.
; This provides a very abstract view of the network.  But it also provides the
; lowermost level of what we call the "routing stack".  Note that at this level
; routing tables are not modified, and system topology is not changed.  Given an
; IP level implementation, this provides the spec for that implementation
; (with appropriate abstract definitions of the stubbed functions).

; We include the records book since we use it to model and specify state
; components.  For fast execution, we use the "fast records" book rather than
; "misc/records".
(include-book "defexec/other-apps/records/records" :dir :system)

(defmacro g (key r)
  `(mget ,key ,r))

(defmacro s (key val r)
  `(mset ,key ,val ,r))

; The included records book doesn't supply a notion of "all the keys of a
; record".  So we define that now.

(defun all-keys (r)
  (strip-cars r))

; We define an enqueue function since we don't want to think about append.
(defund enqueue (e l) (append l (list e)))

; We also need a lastval function.
(defun lastval (x)
  (cond ((endp x) nil)
        ((endp (rest x)) (first x))
        (t (lastval (rest x)))))

; We define memberp only so that we have a membership test that provably
; returns a Boolean, unlike member.
(defun memberp (e x)
  (cond ((endp x) nil)
        ((equal e (first x)) t)
        (t (memberp e (rest x)))))

; A ``type-checking'', ``wild-hair'' idea: Macros like the following could have
; guards that specify that names of their formal parameters are initial
; substrings of corresponding names of their actual parameters, unless a
; keyword :names-ok is supplied (which could be :ALL or could be a list of
; formal parameter names).  More generally, the guard could allow a formal
; parameter p to correspond to an actual parameter (g :pkey ...), where pkey
; has p as an initial substring.

; Here are the state components for a router.
(defmacro pending-queue (r) `(g :pending-queue ,r))
(defmacro local-state   (r) `(g :local-state   ,r))
(defmacro routing-table (r) `(g :routing-table ,r))

; Here are the components of a packet.
(defmacro src           (pkt) `(g :source      ,pkt))
(defmacro destination   (pkt) `(g :destination ,pkt))
(defmacro pkt-content   (pkt) `(g :content     ,pkt))

; Here are the components of an oracle entry.
(defmacro index    (o) `(g :index    ,o))
(defmacro cmd-type (o) `(g :cmd-type ,o))

; Finally here are the state components.
(defmacro routers  (st) `(g :routers ,st))
(defmacro network  (st) `(g :network ,st))

; And here's code for updates.

(defun our-keyword-value-listp (l)

; This is keyword-value-listp except that we allow numeric keys.

  (declare (xargs :guard t))
  (cond ((atom l) (null l))
        (t (and (or (keywordp (car l))
                    (natp (car l)))
                (consp (cdr l))
                (our-keyword-value-listp (cddr l))))))

(defun update-macro (upds result)
  (declare (xargs :guard (our-keyword-value-listp upds)))
  (if (endp upds) result
    (update-macro (cddr upds)
                  (list 's (car upds) (cadr upds) result))))

(defmacro update (old &rest updates)
  (declare (xargs :guard (our-keyword-value-listp updates)))
  (update-macro updates old))

(defmacro >r  (&rest upds) `(update r  ,@upds))
(defmacro >st (&rest upds) `(update st ,@upds))

; We stub out three functions here, which will be modeled as we refine the
; framework.  These have to do with (1) what a router is supposed to do when it
; receives a packet, (2) how a router (which might also be a host) generates a
; packet to be delivered, and (3) how the next hop is computed given a
; destination and a routing table.

; The function consume-pkt updates the local state of a process once it
; consumes a packet that is destined for it.
(encap
 (((consume-pkt * *) => *))
 (defun-loc consume-pkt (pkt local-state)
   (update local-state
           :consumed (cons (pkt-content pkt)
                           (g :consumed local-state)))))

; The function create-pkt creates a packet given a particular local state and a
; routing table.  One might be surprised that this is a function of the local
; state rather than non-deterministically selected from the oracle.  The reason
; is that we are thinking of the current model as the "bottom level" of a
; framework on top of which we can build progressively more sophisticated
; protocols at higher levels (for example BGP and other stuff).  But a BGP
; protocol (built on top of TCP) actually must generate the appropriate packets
; to facilitate creation of BGP sessions and exchange of routing information.
; So we abstract all that in the model of the network level with this stubbed
; function.

; Eventually we may want to break local state into somewhat independent
; pieces, since as things stand now, it will be difficult to state correct
; non-interference properties because local state affects packet
; destinations.

(encap
 (((create-pkt * *) => *))

 (defun-loc create-pkt (local-state routing-table)
   (declare (ignore routing-table))
   (update nil
           :content (g :content local-state)
           :destination (g :destination local-state)
           :source (g :source local-state))))

; The next-hop function determines, given a routing table and a destination id,
; which router to send the packet to.  We will make use of an assumption that
; there is no router corresponding to the key (invalid).

; Note that we might want to return (invalid) if it would be a security
; violation to forward a packet.  In that case next-hop should take the
; contents of a packet, or perhaps the packet itself, as an additional
; argument.  However, we defer such complexity to future versions of this
; model.

(encap
 (((invalid) => *)
  ((next-hop * *) => *))

 (defun-loc invalid () nil)

 (defun-loc next-hop (d rt)
   (or (g d rt)
       (g :default rt)
       (invalid)))

 (defthm |invalid for nil|
   (implies (not rt)
            (equal (next-hop d rt) (invalid)))

; Hints are included for the case #+acl2-exec.

   :hints (("Goal" :in-theory (enable next-hop)))))

; The function update-local-state just updates the local state in some way.
(encap
 (((update-local-state *) => *))
 (defun-loc update-local-state (local-state)
   (let ((counter (or (g :counter local-state)
                      0)))
     (update local-state
             :counter (1+ counter)
             :destination (if (or (equal (g :destination local-state)
                                         0)
                                  (not (natp (g :destination
                                                local-state))))
                              0
                            (1- (g :destination local-state)))
             :content (cons (g :content local-state)
                            (g :content local-state))))))

(defun delete-assoc (key alist)
  (declare (xargs :guard (eqlable-alistp alist)))
  (cond ((endp alist) nil)
        ((eql key (caar alist)) (cdr alist))
        (t (cons (car alist) (delete-assoc key (cdr alist))))))

; The following functions abstract the notions of looking up a router in a
; network and a router joining a network or being removed from it, where the
; network is some form of an adjacency list that stipulates the neighbors for
; each node.  Note that if i is already in the network, then join-network is a
; no-op.
(encap
 (((network-lookup * *) => *)
  ((join-network * *) => *)
  ((delete-from-network * *) => *))

 (defun-loc network-lookup (i network)
   (cdr (assoc i network)))

 (defun-loc join-network (i network)
   (let ((pair (assoc i network)))
     (if pair
         network
       (put-assoc-eql i nil network))))

 (defun-loc delete-from-network (i network)
   (delete-assoc i network)))

; Now here is another little stub.  Given a router index, the stub AS returns
; the index of the autonomous system.  This part is only important in
; developing a routing configuration.
(encap
 (((AS *) => *))
 (defun-loc AS (i) ; just one autonomous system in this example:
   (declare (ignore i))
   0))

; The function route-pkt is the only "interesting" function which handles the
; routing.

(defun route-pkt (i routers)
  (let* (;; We first pick up the router for index i
         (r (g i routers))

         ;; Then look at its pending queue
         (pending-queue (pending-queue r)))

    (if (not pending-queue)

        ;; No pending message to route, so don't bother.
        routers

      (let* (;; Otherwise we pick the first packet from the pending
     ;; queue
; Consider making our picking more generic.  A realistic system can have a
; complex algorithm for constraining which packet to pick.

             (pkt (first pending-queue))

             ;; and remove the packet.
             (pending-queue (rest pending-queue))

             ;; Then update the router
             (r (>r :pending-queue pending-queue))

             ;; Now we look at the destination of the packet.
             (destination (destination pkt)))

        (if (equal destination i)

            ;; The packet is for this router.  So it consumes the packet and
            ;; updates its local state.  Once we get down to it, the routing
            ;; table might be updated as well, but we do it as a separate thing
            ;; for now.

            (let* ((local-state (local-state r))
                   (local-state (consume-pkt pkt local-state))
                   (r (>r :local-state local-state)))
              (s i r routers))
          (let* ((routers (s i r routers))

                 ;; Otherwise it looks up the routing table.
                 (routing-table (routing-table r))

                 ;; determines what the next hop for this packet will
                 ;; be.
                 (next-hop (next-hop destination routing-table)))
            (if (equal next-hop (invalid))
                ;; No path to the host; we just forget it.
                routers

              (let* (;; Otherwise we send off the packet to the next hop.
                     (r (g next-hop routers))
                     (pending-queue (pending-queue r))
                     (pending-queue
                      (enqueue pkt pending-queue))
                     (r (>r :pending-queue pending-queue))
                     (routers (s next-hop r routers)))
                routers))))))))

; The two following functions are completely trivial.  The first function
; generates a packet.  We simply generate the packet based on the current local
; state and routing table.  (The routing table is taken into account since
; presumably it might have a role to play when we build BGP protocols on top of
; this, which send routing messages.  Those messages are presumably created and
; sent to other routers depending on the routing table.)  The second is just a
; way of updating the local state.  We think of this as performing ordinary
; computation.

(defun generate-pkt (i routers)
  (let* ((r (g i routers))
         (local-state (local-state r))
         (routing-table (routing-table r))
         (pkt (create-pkt local-state routing-table))
         (pending-queue (pending-queue r))
         (pending-queue (enqueue pkt pending-queue))
         (r (>r :pending-queue pending-queue)))
    (s i r routers)))

(defun perform-computation (i routers)
  (let* ((r (g i routers))
         (local-state (local-state r))
         (local-state (update-local-state local-state))
         (r (>r :local-state local-state)))
    (s i r routers)))

; Now we go to configuring a router as it joins the network.

(encap
 (((initial-configuration * * *) => *))
 (defun-loc initial-configuration (i AS routers)
   ;; doesn't do much at this point
   (declare (ignore i AS routers))
   nil))

(defun initialize-router (i routers)
  (if (g i routers)
      ;; The router already exists so nothing to configure (for now)
      routers

    ;; Otherwise we assume that there is some initial configuration
    ;; which is actually stipulated by the AS.
    (s i (initial-configuration i (AS i) routers) routers)))

; Finally, we talk about changing a router configuration.  This can happen in a
; number of occasions, which should be governed by an oracle.  For now we just
; stub this.

(encap
 (((update-configuration * * *) => *))
 (defun-loc update-configuration (i AS routers)
   ;; doesn't do much at this point
   (declare (ignore i AS routers))
   nil))
```

```
; Now the system step is a trivial matter.

(defun system-step (st oracle-entry)
  (let* ((routers (routers st))
         (network (network st))
         (i (index oracle-entry))
         (cmd-type (cmd-type oracle-entry)))
    (if (equal i (invalid))
        st
      (case cmd-type
        (:route        (>st :routers (route-pkt i routers)))
        (:generate-pkt (>st :routers (generate-pkt i routers)))
        (:compute      (>st :routers (perform-computation i routers)))
        (:drop-node    (>st :routers (s i nil routers)
                            :network (delete-from-network i network)))
        (:add-node     (>st :network (join-network i network)
                            :routers (initialize-router i routers)))
        (:change-conf  (>st :routers (update-configuration i (AS i) routers)))
        (t st)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section 3: Formalization of network properties: Validity and Visibility
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; What properties can be guaranteed by the network at the level shown in
; Section 2?  Note that the model in Section 2 does not guarantee anything,
; obviously, since it has a bunch of stubbed-out functions.  But the way we are
; thinking about the approach is the following.  We will keep the model (in
; section 2) as abstract as possible, and then build protocols on top of that.
; When we prove properties of those protocols, we *assume* that the underlying
; network provides some generic properties as "eventual invariants".  (We will
; explain what we mean by that in a moment.)  This will let us write properties
; of the higher-level protocols in terms of those eventual invariants.  On the
; other hand, we can define concrete refinements "below" these abstractions, by
; formalizing implementations and showing that the implementations satisfy real
; invariants in order to show that the abstractions satisfying the eventual
; invariants.

; So what do we mean by an "eventual invariant"?  Ideally, we will want to say
; that a property is an invariant of the system, so that when we analyze the
; system we can assume that it always satisfies that property.  But the
; properties of concern here, such as validity and visibility, are not going to
; be invariants of the system (once we have a system with all the features we
; need, for example possible link failures and other such properties).  But
; what we want to say is that they are still invariants if the network is
; stabilized, that is if routing tables stop changing.  (Aside: This reminds me
; of flushing in pipelines.  Many properties of pipelines are not invariants in
; general, but become invariants as we go from flushed states to flushed
; states.  Just an analogy at this point, though.)

; We now focus on formalizing validity.  To do so, we have to understand a bit
; more how the network topology is related to routing tables.

; We view a network as a graph where each node (a router) is connected to a
; bunch of nodes that specify its neighbors.  So we can easily define a path
; along that graph.

(defun pathp (p network)
  (cond  ((endp p)
          t)
         ((endp (rest p))
          (not (equal (first p) (invalid))))
         (t
          (and (not (equal (first p) (invalid)))
               (memberp (second p) (network-lookup (first p) network))
               (pathp (rest p) network)))))

; Now we introduce the notion of an "induced path", that is, a path that is
; created by going successively along the routing tables of the routers, as
; specified.  Note that our routing tables are deterministic, even though in
; the actual Internet, two packets from the same source with the same
; destination might take different first hops.  However, our model reflects how
; this works in the Internet, by updating the routing tables when different
; paths are to be taken, using update-configuration.

(defun router-induced-path (routers s d n)
  (declare (xargs :measure (nfix n)))
  (cond ((zp n) nil)
        ((equal s d) (list s))
        (t (let* ((r (g s routers))
                  (routing-table (routing-table r))
                  (next-hop (next-hop d routing-table)))
             (cons s (router-induced-path routers next-hop d (1- n)))))))

; Now we define validity.  The predicate system-valid says that for each valid
; source-destination pair, the induced path from s to d along the routing table
; corresponds to a physical path in the network.

(defun s-d-valid-matrix (routers network s d n)
  (let ((rp (router-induced-path routers s d n)))
    (and (pathp rp network)
         (equal s (first rp))
         (equal d (lastval rp)))))

(defun-sk s-d-valid (routers network s d)
  (exists n
          (s-d-valid-matrix routers network s d n)))

(defun validity-holds-matrix (routers network s d)

; Keep this in sync with validity-holds-exec-matrix.
```

```
  (let* ((r (g s routers))
         (routing-table (routing-table r)))
    (implies
     (and
      (not (equal s (invalid)))
      (not (equal d (invalid)))
      (not (equal (next-hop d routing-table) (invalid))))
     (s-d-valid routers network s d))))

(defun-sk validity-holds (routers network)
  (forall (s d)
          (validity-holds-matrix routers network s d)))

; We develop here an executable version of validity-holds that can produce a
; counterexample.

; A nice challenge would be to prove equivalence of s-d-valid with the
; following executable version.

(defun s-d-valid-exec (routers network s d)
  (let ((n (length (all-keys routers))))
    (s-d-valid-matrix routers network s d n)))

(defun validity-holds-exec-matrix (routers network s d)

; Keep this in sync with validity-holds-matrix.

  (let* ((r (g s routers))
         (routing-table (routing-table r)))
    (implies
     (and
      (not (equal s (invalid)))
      (not (equal d (invalid)))
      (not (equal (next-hop d routing-table) (invalid))))
     (s-d-valid-exec routers network s d))))

(defun validity-failure-exec-s (routers network s dests)
  (if (endp dests)
      nil
    (if (validity-holds-exec-matrix routers network s (car dests))
        (validity-failure-exec-s routers network s (cdr dests))
      (list s (car dests)))))

(defun validity-failure-exec-s-d (routers network sources dests)
  (if (endp sources)
      nil
    (or (validity-failure-exec-s routers network (car sources) dests)
        (validity-failure-exec-s-d routers network (cdr sources) dests))))

(defun validity-failure-exec (routers network)
  (let ((all-node-ids (strip-cars routers)))
    (validity-failure-exec-s-d routers network all-node-ids
                               all-node-ids)))

(defun validity-holds-exec (routers network)
  (not (validity-failure-exec routers network)))

(defun system-valid (st)
  (validity-holds (routers st) (network st)))

(defun system-failure-exec (st)
  (validity-failure-exec (routers st) (network st)))

(defun system-valid-exec (st)
  (validity-holds-exec (routers st) (network st)))

; Now we turn to visibility.  Visibility says that the routing tables should
; provide some path if there is some valid path from s to d.

(defun exists-s-d-path-matrix (network s d p)
  (and (pathp p network)
       (equal (first p) s)
       (equal (lastval p) d)))

(defun-sk exists-s-d-path (network s d)
  (exists p
          (exists-s-d-path-matrix network s d p)))

; It would be good to define an executable version of exists-s-d-path here, but
; that will involve enumerating (or at least recurring through) all paths.  So
; until then, we are stuck in defining executable versions of the functions
; defined below with defun-sk.

(defun s-d-visibility-n (routers network s d n)
  (implies (exists-s-d-path network s d)
           (let ((rp (router-induced-path routers s d n)))
             (and (not (memberp (invalid) rp))
                  (equal (first rp) s)
                  (equal (lastval rp) d)))))

(defun-sk s-d-visibility (routers network s d)
  (exists n (s-d-visibility-n routers network s d n)))

(defun visibility-property-s-d (routers network s d)
  (implies (and (not (equal s (invalid)))
                (not (equal d (invalid))))
           (s-d-visibility routers network s d)))

(defun-sk visibility-property (routers network)
  (forall (s d)
          (visibility-property-s-d routers network s d)))
```

```
(defun system-visibility (st)
  (visibility-property (routers st) (network st)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section 4: Sample simulations
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun wf-alistp (alist)
  (declare (xargs :guard t))
  (if (consp alist)
      (and (consp (car alist))
           (wf-keyp (caar alist))
           (wf-alistp (cdr alist)))
    (null alist)))

(defun alist-to-record (alist)
  (declare (xargs :guard (wf-alistp alist)
                  :verify-guards nil))
  (if (endp alist)
      nil
    (s (caar alist) (cdar alist) (alist-to-record (cdr alist)))))

(defthm good-map-alist-to-record
  (implies (wf-alistp alist)
           (good-map (alist-to-record alist))))

(verify-guards alist-to-record)

(defun wf-alist-alistp (alist)
  (declare (xargs :guard t))
  (if (consp alist)
      (and (consp (car alist))
           (wf-keyp (caar alist))
           (wf-alistp (cdar alist))
           (wf-alist-alistp (cdr alist)))
    (null alist)))

(defconst-exec *rt-A*
  '((0 (1 . 2)
       (4 . 2)
       (2 . 2)
       (5 . 5)
       (6 . 3)
       (:default . 3))
    (2 (1 . 1)
       (4 . 1)
       (7 . 8)
       (:default . 0))
    (1 (4 . 4)
       (:default . 2))
    (3 (7 . 6)
       (8 . 6)
       (9 . 9)
       (10 . 9)
       (:default . 0))
    (4 (:default . 1))
    (5 (0 . 0)
       (:default . 9))
    (6 (3 . 3)
       (:default . 7))
    (7 (6 . 6)
       (:default . 8))
    (8 (7 . 7)
       (:default . 2))
    (9 (10 . 10)
       (:default . 3))
    (10 (:default . 9))))

(defconst-exec *oracle-A*

; Generate path from 3 to 8:
; (3 6 7 8).

  (list (update nil :index 3 :cmd-type :generate-pkt)
        (update nil :index 3 :cmd-type :route)
        (update nil :index 6 :cmd-type :route)
        (update nil :index 7 :cmd-type :route)
        (update nil :index 8 :cmd-type :route)

; Now let's update the counters a bit

        (update nil :index 2 :cmd-type :compute)
        (update nil :index 2 :cmd-type :compute)
        (update nil :index 3 :cmd-type :compute)

        ))

(defconst-exec *local-state-A*
  (update nil
          :source 3
          :destination 8
          :content 'some-content))

(defun routing-table-to-routers (alist)
  (declare (xargs :guard (wf-alist-alistp alist)))
  (if (endp alist)
      nil
    (s (caar alist)
       (update nil :routing-table
               (alist-to-record (cdar alist)))
       (routing-table-to-routers (cdr alist)))))
```

```
(defconst-exec *routers-A*
  (let* ((routers (routing-table-to-routers *rt-A*))
         (entry3 (g 3 routers)))
    (update routers
            3 (update entry3 :local-state *local-state-A*))))

(defun route-table-to-network (tbl)

; We create the minimal network for which the given routing table makes sense.

  (if (endp tbl)
      nil
    (s (caar tbl)
       (remove-duplicates-eql (strip-cdrs (cdar tbl)))
       (route-table-to-network (cdr tbl)))))

(defconst-exec *network-A*
  (route-table-to-network *rt-A*))

(defconst-exec *state-A-0*
  (update nil
          :routers *routers-A*
          :network *network-A*))

(defthm-exec system-valid-exec-test-A
  (equal (system-valid-exec *state-A-0*)
         nil))

(defthm-exec system-failure-exec-test-A
  (equal (system-failure-exec *state-A-0*)
         '(0 6)))

(defthm-exec system-failure-exec-test-A-witness

; We have a loop:

  (pathp '(0 3 0 3 0 3)
         *network-A*))

(defconst-exec *state-A-1*
  (system-step *state-A-0*
               (nth 0 *oracle-A*)))

(defconst-exec *state-A-2*
  (system-step *state-A-1*
               (nth 1 *oracle-A*)))

(defconst-exec *state-A-3*
  (system-step *state-A-2*
               (nth 2 *oracle-A*)))
```

```
(defconst-exec *state-A-4*
  (system-step *state-A-3*
               (nth 3 *oracle-A*)))

(defconst-exec *state-A-5*
  (system-step *state-A-4*
               (nth 4 *oracle-A*)))

(defconst-exec *state-A-6*
  (system-step *state-A-5*
               (nth 5 *oracle-A*)))

(defconst-exec *state-A-7*
  (system-step *state-A-6*
               (nth 6 *oracle-A*)))

(defconst-exec *state-A-8*
  (system-step *state-A-7*
               (nth 7 *oracle-A*)))

(defthm-exec state-8-check
  (equal *state-A-8*
         '((:NETWORK (0 2 5 3)
                     (1 4 2)
                     (2 1 8 0)
                     (3 6 9 0)
                     (4 1)
                     (5 0 9)
                     (6 3 7)
                     (7 6 8)
                     (8 7 2)
                     (9 10 3)
                     (10 9))
           (:ROUTERS (0 (:ROUTING-TABLE (1 . 2)
                                        (2 . 2)
                                        (4 . 2)
                                        (5 . 5)
                                        (6 . 3)
                                        (:DEFAULT . 3)))
                     (1 (:ROUTING-TABLE (4 . 4) (:DEFAULT . 2)))
                     (2 (:LOCAL-STATE (:CONTENT (NIL) NIL)
                                      (:COUNTER . 2)
                                      (:DESTINATION . 0))
                        (:ROUTING-TABLE (1 . 1)
                                        (4 . 1)
                                        (7 . 8)
                                        (:DEFAULT . 0)))
                     (3 (:LOCAL-STATE (:CONTENT SOME-CONTENT . SOME-CONTENT)
                                      (:COUNTER . 1)
                                      (:DESTINATION . 7)
                                      (:SOURCE . 3))
                        (:ROUTING-TABLE (7 . 6)
                                        (8 . 6)
                                        (9 . 9)
                                        (10 . 9)
                                        (:DEFAULT . 0)))
                     (4 (:ROUTING-TABLE (:DEFAULT . 1)))
                     (5 (:ROUTING-TABLE (0 . 0) (:DEFAULT . 9)))
                     (6 (:ROUTING-TABLE (3 . 3) (:DEFAULT . 7)))
                     (7 (:ROUTING-TABLE (6 . 6) (:DEFAULT . 8)))
                     (8 (:LOCAL-STATE (:CONSUMED SOME-CONTENT))
                        (:ROUTING-TABLE (7 . 7) (:DEFAULT . 2)))
                     (9 (:ROUTING-TABLE (10 . 10)
                                        (:DEFAULT . 3)))
                     (10 (:ROUTING-TABLE (:DEFAULT . 9)))))))

(defun run (init-state oracle-list)
  (if (endp oracle-list)
      init-state
    (run (system-step init-state (car oracle-list))
         (cdr oracle-list))))

(defthm-exec run-test-A
  (and (equal *state-A-0*
              (run *state-A-0* (take 0 *oracle-A*)))
       (equal *state-A-3*
              (run *state-A-0* (take 3 *oracle-A*)))
       (equal *state-A-5*
              (run *state-A-0* (take 5 *oracle-A*)))
       (equal *state-A-8*
              (run *state-A-0* (take 8 *oracle-A*)))))

; Let's try again, with x-y routing on a square this time.

; 0 -- 1 -- 2
; |    |    |
; 3 -- 4 -- 5

(defconst-exec *rt-B*
  '((0 (1 . 1)
       (2 . 1)
       (3 . 3)
       (4 . 1)
       (5 . 1))
    (1 (0 . 0)
       (2 . 2)
       (3 . 0)
       (4 . 4)
       (5 . 2))
    (2 (0 . 1)
       (1 . 1)
       (3 . 1)
       (4 . 1)
       (5 . 5))
    (3 (0 . 0)
       (1 . 4)
       (2 . 4)
       (4 . 4)
       (5 . 4))
    (4 (0 . 3)
       (1 . 1)
       (2 . 5)
       (3 . 3)
       (5 . 5))
    (5 (0 . 4)
       (1 . 4)
       (2 . 2)
       (3 . 4)
       (4 . 4))))

(defconst-exec *routers-B*
  (routing-table-to-routers *rt-B*))

(defconst-exec *network-B*
  (route-table-to-network *rt-B*))

(defconst-exec *state-B-0*
  (update nil
          :routers *routers-B*
          :network *network-B*))

(defthm-exec system-valid-exec-test-B
  (equal (system-valid-exec *state-B-0*)
         t))

(defthm-exec system-failure-exec-test-B
  (equal (system-failure-exec *state-B-0*)
         nil))

; A nice thing to do next would be to add node 6:

; 0 -- 1 -- 2
; |    |    |
; 3 -- 4 -- 5
; |
; 6

; But first, we need more interesting witness functions for
; initial-configuration and update-configuration (and perhaps other support not
; yet foreseen).

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section 5: Defining a routing policy
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; We now discuss a formalization of a routing policy.  The policy we'll
; formalize is that every packet from an AS(i) to AS(j) is transmitted along
; AS(k).

; First we define the notion of "border" of AS(i).

(defun-sk exists-next-hop (r routing-table)
  (exists d (equal r (next-hop d routing-table))))

; The following predicate says that r is directly connected to AS(j).

(defun-sk in-border (j r routers)
  (exists rprime
          (and (equal (AS rprime) j)
               (not (equal j (AS r)))
               (let* ((r-state (g r routers))
                      (routing-table (routing-table r-state)))
                 (exists-next-hop rprime routing-table)))))

; The following predicate says that path is a path that takes us from AS(i)
; directly to AS(j).

(defun-sk path-through-AS-k (routers path d i j)
  (exists p
          (and (memberp p path)
               (in-border j p routers)
               (equal (AS p) i)
               (let* ((r-state (g p routers))
                      (r-table (routing-table r-state)))
                 (equal (AS (next-hop d r-table)) j)))))

; Now we want to say that each path of length n from a router in AS(i) to a
; router in AS(k) goes through AS(j).  So we say that suppose the destination
; of the pkt is for k then for each router induced path of length n that takes
; us to the destination d, there must exist a border router p of AS(i) and the
; next-hop for p must be a router in AS(j).

(defun-sk routing-policy-for-n (k j routers n i)
  (forall
   (r pkt)
   (implies
    (and
     (equal i (AS r))
     (equal (AS (destination pkt)) k))
    (let* ((d (destination pkt))
           (path (router-induced-path routers r d n)))
      (implies (and (not (equal d (invalid)))
                    (equal (lastval path) d))
               (path-through-AS-k routers path d i j))))))
```

B Book for Security Properties

The book below is divided into sections, as follows.

Section 1: Generic functions, macros, and state components
Section 2: State transitions
Section 3: Information flow specification functions
Section 4: Information Flow theorem
Section 5: Non-interference theorem
Section 6: Corollaries for initial states

Note that Theorems 2 and 1 above may be found in Section 6, "Corollaries for initial states," as corollaries of stronger theorems proved using suitable inductive invariants.

```
==========

(in-package "ACL2")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section 1: Generic functions, macros, and state components
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(include-book "misc/records" :dir :system)

; The following is redundant when  we include misc/records.  If we include
; misc/records0, we'll need it, but then we of course won't need the enable
; that follows it.

(defthm g-of-s-redux
  (equal (g a (s b v r))
         (if (equal a b) v (g a r))))
(in-theory (enable g-of-s-redux))

(defun update-macro (upds result)
  (declare (xargs :guard (keyword-value-listp upds) :mode :program))
  (if (endp upds) result
    (update-macro (cddr upds)
                  (list (packn (list 'set- (car upds))) (cadr upds) result))))

(defmacro update (old &rest updates)
  (declare (xargs :guard (keyword-value-listp updates)))
  (update-macro updates old))

(defun enqueue (item queue)
  (declare (xargs :measure (acl2-count queue)))
  (if (endp queue)
      (list item)
    (cons (first queue)
  (enqueue item (rest queue)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Technical point:
```

```
; We are about to introduce accessor functions.  Normally we would strongly
; consider defining these as macros instead.  However, we plan to use
; congruence-based reasoning so that ACL2 will automatically replace one
; variable with another assumed node= to it.  Congruence rules are happy to
; take us from preserving an equivalence relation to preserving another when
; diving to the kth argument of any call of a given function, f, e.g.:

; (defcong node= same-len (incoming-packets node) 1)

; But ACL2 doesn't support the use of constant arguments in congruence rules,
; so for example the following is illegal -- yet this is what we would need if
; incoming-packets were a macro.

; (defcong node= same-len (g :incoming-packets node) 2)

(deflabel start-accessors)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; The system state consists of three components:
; 1) a collection of "nodes", and
; 2) and a natural number counter indicating the id for the next packet.

; A node can be a router or a host (or can act as both).

(defund nodes            (st)  (g :nodes           st))
(defund next-packet-id   (st)  (g :next-packet-id  st))

; The following are needed for when update is used.

(defmacro set-nodes (nodes st)
  `(s :nodes ,nodes ,st))

(defmacro set-next-packet-id (id st)
  `(s :next-packet-id ,id ,st))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; A node has the following components:

; 1) a queue of incoming packets.  For a host, these would presumably all be
; for the host.  For a router, these would consist of a mixture of packets to
; be routed or packets for the router.

; 2) a queue of outgoing packets.  These are packets waiting to be forwarded.
; For a host, these would be passed on to the attached router.  For a router,
; these would be sent to an attached host or passed on the the next router in
; the packet's path.

; 3) a routing table indicating where to send outgoing packets.
; 4) a local state for computation and generation of packets.

; and finally, for tracking information flow:

; 5) a "marked" bit (no macro defined for that).

(defund incoming-packets  (node)  (g :incoming-packets node))
(defund outgoing-packets  (node)  (g :outgoing-packets node))
(defund routing-table     (node)  (g :routing-table    node))
(defund local-state       (node)  (g :local-state      node))
(defund marked            (node)  (g :marked           node))

(defund set-incoming-packets  (pkts node)  (s :incoming-packets pkts node))
(defund set-outgoing-packets  (pkts node)  (s :outgoing-packets pkts node))
(defund set-routing-table     (rt node)    (s :routing-table    rt   node))
(defund set-local-state       (ls node)    (s :local-state      ls   node))
(defund set-marked            (flg node)   (s :marked           flg  node))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; What does a packet look like?  A packet has fields including an id, a source,
; a destination, and its contents, as well as its "marked" bit for tracking
; information flow (no macro defined for that).

(defund id           (pkt)  (g :id          pkt))
(defund source       (pkt)  (g :source      pkt))
(defund destination  (pkt)  (g :destination pkt))
(defund contents     (pkt)  (g :contents    pkt))
; (defund marked            (p)  (g :marked           p))

(defund set-id           (id pkt)  (s :id          id pkt))
(defund set-source       (id pkt)  (s :source      id pkt))
(defund set-destination  (id pkt)  (s :destination id pkt))
(defund set-contents     (x pkt)   (s :contents    x  pkt))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Now we specify some fields of an input from the oracle.  The :node-id field
; says which node is to take a step.  The :operation field says what kind of
; step it is, namely computation, packet creation, packet consumption, or
; packet forwarding.  The :dest field provides the destination field of a new
; packet if one is created.

(defund node-id    (input)  (g :node-id   input))
(defund operation  (input)  (g :operation input))
(defund dest       (input)  (g :dest      input))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(deftheory accessors
  (set-difference-theories (function-theory :here)
                           (function-theory 'start-accessors)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section 2: State transitions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Here we define our state transition.

; We start by stubbing out a few functions:

(defstub perform-computation  (local-state)         t)  ;local-state
(defstub consume-packet       (packet local-state)  t)  ;local-state
(defstub update-sent          (local-state)         t)  ;local-state
(defstub find-contents        (local-state)         t)  ;contents

(defund system-step-compute (st node-id)
  (let* ((nodes     (nodes st))
 (node      (g node-id nodes)))
    (let* ((local-state  (local-state node))
           (local-state  (perform-computation local-state))
           (node         (update node :local-state local-state))
           (nodes        (s node-id node nodes))
           (st           (update st :nodes nodes)))
      st)))

(defund new-packet (node-id dest next-pkt-id nodes)

; This function is one of two that deal directly with a :marked field.

  (let* ((node             (g node-id nodes))
         (local-state      (local-state node))
         (contents         (find-contents local-state))
         (marked           (marked node))
         (pkt              (update nil
                                   :id          next-pkt-id
                                   :source      node-id
                                   :destination dest
                                   :contents    contents
                                   :marked      marked)))
    pkt))

(defund system-step-create (st node-id pkt-dest)
  (let* ((nodes     (nodes st))
 (node      (g node-id nodes)))
    (let* ( ;; we first creates the appropriate packet.
           (local-state      (local-state node))
           (next-pkt-id      (next-packet-id st))
           ;; we add it to the list of packets sent.
           (pkt              (new-packet node-id pkt-dest next-pkt-id nodes))
           (st               (update st
                                  :next-packet-id (+ 1 next-pkt-id)))
           (local-state      (update-sent local-state))

           ;; and add it to the outgoing packets queue.
           (outgoing-packets (outgoing-packets node))
           (outgoing-packets (enqueue pkt outgoing-packets))
           (node             (update node
                                     :outgoing-packets outgoing-packets
                                     :local-state local-state))
           (nodes            (s node-id node nodes))
           (st               (update st :nodes nodes)))
      st)))

(defund node-consume-packet (pkt node)

; This function is one of two that deal directly with a :marked field.

  (let* ((local-state (local-state node))
         (local-state (consume-packet pkt local-state))
         (marked-pkt (marked pkt))
         (node (if (and marked-pkt
                        (not (marked node)))
                   (update node
                           :local-state local-state
                           :marked marked-pkt)
                 (update node
                         :local-state local-state))))
    node))

(defund system-step-consume (st node-id)
  (let* ((nodes     (nodes st))
 (node      (g node-id nodes)))
    (let ((incoming-packets (incoming-packets node)))
      (if (endp incoming-packets)
           ;; There are no incoming packets, so there is nothing to do.
           st
         ;; Otherwise we pop the first packet
         (let* ((pkt  (first incoming-packets))
                (node (update node :incoming-packets (rest incoming-packets)))
                (dest (destination pkt)))

           (if (equal dest node-id)
               ;; packet is for me.
               (let* ((node (node-consume-packet pkt node))
                      (nodes (s node-id node nodes))
                      (st (update st :nodes nodes)))
                 st)
             ;; packet is for someone else. Presumably we are a router,
             ;; so we move the packet to the outgoing packets queue
             ;; for subsequent forwarding.
             (let* ((outgoing-packets (outgoing-packets node))
                 (outgoing-packets (enqueue pkt outgoing-packets))
                 (node             (update node :outgoing-packets
                                           outgoing-packets))
                 (nodes            (s node-id node nodes))
                 (st               (update st :nodes nodes)))
             st)))))))

(defund system-step-forward (st node-id)
  (let* ((nodes     (nodes st))
 (node      (g node-id nodes)))
    (let ((outgoing-packets (outgoing-packets node)))
      ;; If there's no packet, there's nothing to do.
      (if (endp outgoing-packets)
           st
         (let* ( ;; Otherwise we pop the first packet.
                (pkt              (first outgoing-packets))
                (outgoing-packets (rest outgoing-packets))
                (node             (update node :outgoing-packets
                                          outgoing-packets))
                (nodes            (s node-id node nodes))

                ;; Then it determines its one-hop destination.
                ;; If it is delivering to a directly attached host,
                ;; this will be the hosts address.  Otherwise it
                ;; will be the next router in the path.
                (pkt-dest         (destination pkt))
                (routing-table    (routing-table node))
                (one-hop-dest     (g pkt-dest routing-table)))
           (let* (
                 ;; and updates the appropriate incoming packets queue.
                 (one-hop-node     (g one-hop-dest nodes))
                 ;; Note: one-hop-node could be nil.  If we imagine that there
                 ;; is a nil node-id -- think of it as a sort of default --
                 ;; then everything should work out.
                 (one-hop-queue    (incoming-packets one-hop-node))
                 (one-hop-queue    (enqueue pkt one-hop-queue))
                 (one-hop-node     (update one-hop-node
                                           :outgoing-packets
                                           one-hop-queue))
                 (nodes            (s one-hop-dest one-hop-node nodes))

                 (st               (update st :nodes nodes)))
             st))))))

(defund system-step (st input)
  (let ((node-id (node-id input)))
    (case (operation input)
      (:compute
       ;; A computation changes the local state of the node.
       (system-step-compute st node-id))
      (:create
       ;; We create a packet and place it on node's outgoing packet
       ;; queue.  We also add the packet to the state's history list
       ;; of all packets created.
       (system-step-create st node-id (dest input)))
      (:consume
       ;; If a packet is for node, we update node's local state to
       ;; reflect its reception.  Otherwise (presumably node is a
       ;; router) we move it to node's outgoing packet queue.
       (system-step-consume st node-id))
      (:forward
       ;; Presumably node is a router.
       (system-step-forward st node-id))
      (t
       st))))

(defun run (st oracle)
  (if (endp oracle)
      st
    (run (system-step st (first oracle))
         (rest oracle))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section 3: Information flow specification functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defund erase-local-state (node)
  (if (marked node)
      (set-local-state nil node)
    node))

(defund erase-packet (pkt)
  (if (marked pkt)
      (set-contents nil pkt)
    pkt))

(defun erase-packets (pkts)
  (if (endp pkts)
      nil
    (cons (erase-packet (car pkts))
          (erase-packets (cdr pkts)))))

(defund erase-incoming-queues (node)
  (set-incoming-packets (erase-packets (incoming-packets node)) node))

(defund erase-outgoing-queues (node)
  (set-outgoing-packets (erase-packets (outgoing-packets node)) node))

(defund erase-node (node)
  (erase-local-state
   (erase-incoming-queues
    (erase-outgoing-queues
     node))))

(defmacro defun-equiv (name arglist &rest rest)
  `(progn (defun ,name ,arglist ,@rest)
          (defequiv ,name)
          (in-theory (disable ,name))))

(defun-equiv node= (node1 node2)
  (equal (erase-node node1)
         (erase-node node2)))

(defcong node= equal (erase-node node) 1
  :hints (("Goal" :in-theory (enable node=))))

(defun-sk nodes= (nodes1 nodes2)
  (forall node-id
          (node= (g node-id nodes1) (g node-id nodes2)))
  :rewrite :direct)

(in-theory (disable nodes=))

(encapsulate
 ()

 (local
  (defthm nodes=-reflexive
    (nodes= x x)
    :hints (("Goal" :in-theory (enable nodes=)))))

 (local
  (defthm nodes=-symmetric
    (implies (nodes= x y)
             (nodes= y x))
    :hints (("Goal" :expand ((nodes= y x))))))

 (local
  (defthm nodes=-transitive
    (implies (and (nodes= x y)
                  (nodes= y z))
             (nodes= x z))
    :hints (("Goal" :expand ((nodes= x z))))))

 (defequiv nodes=))

(in-theory (disable nodes=-necc)) ; loops

(defmacro enable-theory (name &optional names)
  `(union-theories (theory ',name)
                   ,(if names
                        `(enable ,@names)
                      '(current-theory :here))))
```

```
(defun-equiv st= (st1 st2)
  (and (equal (next-packet-id st1) (next-packet-id st2))
       (nodes= (nodes st1) (nodes st2))))

(defun-equiv pkt= (pkt1 pkt2)
  (equal (erase-packet pkt1)
         (erase-packet pkt2)))

(defun-equiv pkts= (pkts1 pkts2)
  (equal (erase-packets pkts1)
         (erase-packets pkts2)))

; Start proof of (defcong node= pkts= (incoming-packets node) 1)

(defthm incoming-packets-erase-node
  (equal (incoming-packets (erase-node node))
         (erase-packets (incoming-packets node)))
  :hints (("Goal"
           :in-theory
           (enable-theory accessors
                          (erase-node
                           erase-local-state
                           erase-incoming-queues
                           erase-outgoing-queues)))))

(defthm erase-packets-erase-packets
  (equal (erase-packets (erase-packets pkts))
         (erase-packets pkts))
  :hints (("Goal" :in-theory (enable-theory accessors
                                            (erase-packets erase-packet)))))

(defcong node= pkts= (incoming-packets node) 1
  :hints (("Goal"
           :in-theory (enable node= pkts=)
           :use
           ((:instance
             (:theorem
              (implies
               (equal node1 node2)
               (equal (erase-packets (incoming-packets node1))
                      (erase-packets (incoming-packets node2)))))
             (node1 (erase-node node))
             (node2 (erase-node node-equiv)))))))

(defthm outgoing-packets-erase-packets
  (equal (outgoing-packets (erase-node node))
         (erase-packets (outgoing-packets node)))
  :hints (("Goal"
           :in-theory
           (enable-theory accessors
                         (erase-node
                          erase-local-state
                          erase-incoming-queues
                          erase-outgoing-queues)))))

(defcong node= pkts= (outgoing-packets node) 1
  :hints (("Goal"
           :in-theory (enable node= pkts=)
           :use
           ((:instance
             (:theorem
              (implies
               (equal node1 node2)
               (equal (erase-packets (outgoing-packets node1))
                      (erase-packets (outgoing-packets node2)))))
             (node1 (erase-node node))
             (node2 (erase-node node-equiv)))))))

; Start proof of (defcong node= equal (routing-table node) 1)

(defthm routing-table-erase-node
  (equal (routing-table (erase-node node))
         (routing-table node))
  :hints (("Goal" :in-theory (enable-theory accessors
                                            (erase-node
                                             erase-local-state
                                             erase-incoming-queues
                                             erase-outgoing-queues)))))

(defcong node= equal (routing-table node) 1
  :hints (("Goal"
           :use
           ((:instance
             (:theorem
              (implies (equal node1 node2)
                       (equal (routing-table node1)
                              (routing-table node2))))
             (node1 (erase-node node))
             (node2 (erase-node node-equiv)))))))

; Start proof of (defcong node= equal (marked node) 1)

(defthm marked-erase-node
  (equal (marked (erase-node node))
         (marked node))
  :hints (("Goal" :in-theory (enable-theory accessors
                                            (erase-node
                                             erase-local-state
                                             erase-incoming-queues
                                             erase-outgoing-queues)))))
```

```
(defcong node= equal (marked node) 1
  :hints (("Goal"
           :use
           ((:instance
             (:theorem
              (implies (equal node1 node2)
                       (equal (marked node1)
                              (marked node2))))
             (node1 (erase-node node))
             (node2 (erase-node node-equiv)))))))

(defcong nodes= node= (g node-id nodes) 2
  :hints (("Goal"
           :in-theory (enable nodes=-necc)
           :restrict ((nodes=-necc ((nodes1 nodes)))))))

(defthm nodes-s-nodes
  (equal (nodes (s :nodes ns st))
         ns)
  :hints (("Goal" :in-theory (enable nodes))))

(defun-equiv consp= (x y)
  (equal (consp x) (consp y)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section 4: Information Flow theorem
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Start proof of node=-step-compute.

(defthm s-local-state-no-op
  (implies (marked node)
           (node= (set-local-state ls node)
                  node))
  :hints (("Goal" :in-theory (enable-theory accessors
                                             (node= erase-node
                                              erase-local-state
                                              erase-incoming-queues
                                              erase-outgoing-queues)))))

(defthm nodes=-forward
  (implies (and (nodes= nodes1 nodes2)
                (syntaxp (not (term-order nodes2 nodes1))))
           (node= (g node-id nodes2)
                  (g node-id nodes1)))
  :rule-classes ((:forward-chaining :trigger-terms ((g node-id nodes2)))))

; Start proof of not-marked-implies-same-local-state

(defthmd local-state-erase-node
  (equal (local-state (erase-node node))
         (if (marked node)
             nil
           (local-state node)))
  :hints (("Goal"
           :in-theory
           (enable-theory accessors
                          (node= erase-node
                                 erase-local-state
                                 erase-incoming-queues
                                 erase-outgoing-queues)))))

; ACL2 proof note: The following would be naturally expressed as a conditional
; congruence rule: we can preserve node= when diving into local-state, provided
; the node is not marked.
(defthm not-marked-implies-same-local-state
  (implies (and (not (marked node1))
                (node= node1 node2)
                (syntaxp (not (term-order node2 node1))))
           (equal (local-state node2)
                  (local-state node1)))
  :hints (("Goal"
           :use ((:instance
                  (:theorem
                   (implies (equal node1 node2)
                            (equal (local-state node2)
                                   (local-state node1))))
                  (node1 (erase-node node1))
                  (node2 (erase-node node2)))
                 (:instance local-state-erase-node
                            (node node1))
                 (:instance local-state-erase-node
                            (node node2))))))

; Start proof of s-local-state-preserves-node=.

(defthm s-local-state-preserves-node=-marked
  (implies (and (node= x x-equiv)
                (marked x)
                (marked x-equiv))
           (equal (node= (set-local-state ls x)
                         (set-local-state ls x-equiv))
                  t))
  :hints (("Goal"
           :in-theory
           (enable-theory accessors
                          (node= erase-node
                                 erase-local-state
                                 erase-incoming-queues
                                 erase-outgoing-queues))))
  :rule-classes nil)
```

```
(defthm s-local-state-preserves-node=-unmarked
  (implies (and (node= x x-equiv)
                (not (marked x))
                (not (marked x-equiv)))
           (equal (node= (set-local-state ls x)
                         (set-local-state ls x-equiv))
                  t))
  :hints (("Goal"
           :in-theory
           (enable-theory accessors
                          (node= erase-node
                                 erase-local-state
                                 erase-incoming-queues
                                 erase-outgoing-queues))))
  :rule-classes nil)

(defcong node= node= (set-local-state ls x) 2
  :hints (("Goal" :use (s-local-state-preserves-node=-unmarked))))

(defthm node=-step-compute
  (implies (and (st= st1 st2)
                (syntaxp (not (term-order st2 st1))))
           (node= (g any-node-id
                     (nodes (system-step-compute st2 node-id)))
                  (g any-node-id
                     (nodes (system-step-compute st1 node-id)))))
  :hints (("Goal" :in-theory (enable system-step-compute st=)
           :cases ((marked (g any-node-id (nodes st1)))))))

; Start proof of node=-step-consume

(defcong pkts= equal (consp pkts) 1
  :hints (("Goal" :in-theory (enable pkts=))))

; Not needed, since we already have equal to pkts= to node=:
; (defcong node= consp= (incoming-packets node) 1
;   :hints (("Goal" :in-theory (enable consp=))))

(defthm node-consume-pkt-no-op-for-marked
  (implies (marked node)
           (node= (node-consume-packet pkt node)
                  node))
  :hints (("Goal" :in-theory (enable node-consume-packet))))

(defcong pkts= pkts= (cdr pkts) 1
  :hints (("Goal" :in-theory (enable pkts=))))

; Start proof of s-incoming-pkts-preserves-node=

(defthm s-incoming-pkts-erase-node
  (equal (erase-node (set-incoming-packets pkts node))
         (set-incoming-packets (erase-packets pkts) (erase-node node)))
  :hints (("Goal"
           :in-theory
           (enable-theory accessors
                          (node= erase-node
                                 erase-local-state
                                 erase-incoming-queues
                                 erase-outgoing-queues)))))

(defcong pkts= pkt= (car pkts) 1
  :hints (("Goal" :in-theory (enable pkts= pkt=))))

(defcong pkt= equal (destination pkt) 1
  :hints (("Goal" :in-theory (enable-theory accessors
                                            (pkt= destination erase-packet))
           :use ((:instance
                  (:theorem
                   (implies (equal pkt1 pkt2)
                            (equal (destination pkt1)
                                   (destination pkt2))))
                  (pkt1 (set-contents nil pkt))
                  (pkt2 (set-contents nil pkt-equiv)))))))

; Start proof of pkt=-implies-node=-node-consume-packet-1-case-1

(defthm marked-set-local-state
  (equal (marked (set-local-state ls node))
         (marked node))
  :hints (("Goal" :in-theory (enable-theory accessors))))

(defthm marked-set-contents
  (equal (marked (set-contents c pkt))
         (marked pkt))
  :hints (("Goal" :in-theory (enable-theory accessors))))

(defthm set-marked-set-local-state
  (implies marked
           (node= (set-marked marked (set-local-state ls node))
                  (set-marked marked node)))
  :hints (("Goal"
           :in-theory
           (enable-theory accessors
                          (node= erase-node
                                 erase-local-state
                                 erase-incoming-queues
                                 erase-outgoing-queues)))))

(encapsulate
 ()

 (local
  (defthm marked-same-for-same-set-contents-lemma
    (equal (equal (g :marked (s :contents c2 pkt2))
                  (g :marked (s :contents c1 pkt1)))
           (equal (g :marked pkt2)
                  (g :marked pkt1)))
    :rule-classes nil))

 (defthm marked-same-for-same-set-contents
   (implies (and (equal (set-contents c1 pkt1)
                        (set-contents c2 pkt2))
                 (syntaxp (not (term-order pkt2 pkt1))))
            (equal (marked pkt2)
                   (marked pkt1)))
   :hints (("Goal" :in-theory (enable-theory accessors)
            :use marked-same-for-same-set-contents-lemma))))

(defcong pkt= node= (node-consume-packet pkt node) 1
  :hints (("Goal" :in-theory (enable node-consume-packet erase-packet pkt=)
           :cases ((marked (node-consume-packet pkt node))))))

(defthm node=-set-marked

; ACL2 proof note: This would make a nice conditional congruence rule,
; conditioned on marked being true.

  (implies (and (node= node1 node2)
                (syntaxp (not (term-order node2 node1)))
                (not (marked node1))
                m)
           (node= (set-marked m node2)
                  (set-marked m node1)))
  :hints (("Goal"
           :in-theory
           (enable-theory accessors
                          (node= erase-node
                                 erase-local-state
                                 erase-incoming-queues
                                 erase-outgoing-queues)))))

(encapsulate
 ()

 (local
  (defthm lemma
    (implies (and (node= node node-equiv)
                  (syntaxp (not (term-order node-equiv node)))
                  (not (marked node))
                  (not (marked node-equiv)))
             (node= (node-consume-packet pkt node)
                    (node-consume-packet pkt node-equiv)))
    :hints (("Goal"
            :in-theory (enable node-consume-packet)))))

 (defcong node= node= (node-consume-packet pkt node) 2
   :hints (("Goal"
            :cases ((marked node))))))

(defcong pkts= node= (set-incoming-packets pkts node) 1
  :hints (("Goal" :in-theory (enable node= pkts=))))

(defcong node= node= (set-incoming-packets pkts node) 2
  :hints (("Goal" :in-theory (enable node=))))

(defthm s-outgoing-pkts-erase-node
  (equal (erase-node (set-outgoing-packets pkts node))
         (set-outgoing-packets (erase-packets pkts) (erase-node node)))
  :hints (("Goal"
           :in-theory
           (enable-theory accessors
                          (node= erase-node
                                 erase-local-state
                                 erase-incoming-queues
                                 erase-outgoing-queues)))))

(defcong pkts= node= (set-outgoing-packets pkts node) 1
  :hints (("Goal" :in-theory (enable node= pkts=))))

(defcong node= node= (set-outgoing-packets pkts node) 2
  :hints (("Goal" :in-theory (enable node=))))

(defcong pkt= pkts= (enqueue pkt pkts) 1
  :hints (("Goal" :in-theory (enable pkts= pkt=))))

(defun double-cdr-induction (x y)
  (if (and (consp x) (consp y))
      (double-cdr-induction (cdr x) (cdr y))
    (list x y)))

(defcong pkts= pkts= (enqueue pkt pkts) 2
  :hints (("Goal" :in-theory (enable pkts=)
           :induct (double-cdr-induction pkts pkts-equiv))))

(defthm node=-step-consume
  (implies (and (st= st1 st2)
                (syntaxp (not (term-order st2 st1))))
           (node= (g any-node-id
                     (nodes (system-step-consume st2 node-id)))
                  (g any-node-id
                     (nodes (system-step-consume st1 node-id)))))
  :hints (("Goal" :in-theory (enable system-step-consume st=))))

; Start proof of node=-step-create.
```

```
(defthm marked-set-outgoing-packets
  (equal (marked (set-outgoing-packets pkts node))
         (marked node))
  :hints (("Goal" :in-theory (enable-theory accessors))))

; Start proof of
; (defcong nodes= pkt= (new-packet node-id dest next-pkt-id nodes) 4)

(encapsulate
 ()

 (local
  (defthm nodes=-implies-pkt=-new-packet-4-marked
    (implies (and (nodes= nodes nodes-equiv)
                  (marked (g node-id nodes)))
             (pkt= (new-packet node-id dest next-pkt-id nodes)
                   (new-packet node-id dest next-pkt-id nodes-equiv)))
    :hints (("Goal" :in-theory (enable-theory accessors
                                              (erase-packet pkt= new-packet))))
    :rule-classes nil))

 (local
  (defthm nodes=-implies-pkt=-new-packet-4-not-marked
    (implies (and (nodes= nodes nodes-equiv)
                  (not (marked (g node-id nodes))))
             (pkt= (new-packet node-id dest next-pkt-id nodes)
                   (new-packet node-id dest next-pkt-id nodes-equiv)))
    :hints (("Goal" :in-theory (enable new-packet)))
    :rule-classes nil))

 (defcong nodes= pkt= (new-packet node-id dest next-pkt-id nodes) 4
   :hints (("Goal" :use (nodes=-implies-pkt=-new-packet-4-marked
                         nodes=-implies-pkt=-new-packet-4-not-marked)))))

(defthm node=-step-create
  (implies (and (st= st1 st2)
                (syntaxp (not (term-order st2 st1))))
           (node= (g any-node-id
                     (nodes (system-step-create st2 node-id dest)))
                  (g any-node-id
                     (nodes (system-step-create st1 node-id dest)))))
  :hints (("Goal" :in-theory (enable system-step-create st=)
           :cases ((marked (g any-node-id (nodes st1)))))))

; Now the proof of node=-step-forward completes automatically, presumably using
; congruence rules already developed above.

(defthm node=-step-forward
  (implies (and (st= st1 st2)
                (syntaxp (not (term-order st2 st1))))
           (node= (g any-node-id
                     (nodes (system-step-forward st2 node-id)))
                  (g any-node-id
                     (nodes (system-step-forward st1 node-id)))))
  :hints (("Goal" :in-theory (enable system-step-forward st=))))

(defcong st= nodes= (nodes st) 1
  :hints (("Goal" :in-theory (enable st=))))

(defthm node=-step
  (implies (and (st= st1 st2)
                (syntaxp (not (term-order st2 st1))))
           (node= (g node-id (nodes (system-step st2 input)))
                  (g node-id (nodes (system-step st1 input)))))
  :hints (("Goal" :in-theory (enable system-step))))

(defthm nodes=-step
  (implies (and (st= st1 st2)
                (syntaxp (not (term-order st2 st1))))
           (nodes= (nodes (system-step st2 input))
                   (nodes (system-step st1 input))))
  :hints (("Goal" :in-theory (enable nodes=))))

(defthm next-packet-id-system-step
  (implies (and (st= st1 st2)
                (syntaxp (not (term-order st2 st1))))
           (equal (next-packet-id (system-step st2 input))
                  (next-packet-id (system-step st1 input))))
  :hints (("Goal" :in-theory
           (union-theories (theory 'accessors)
                           (enable st=
                                   system-step
                                   system-step-compute
                                   system-step-consume
                                   system-step-create
                                   system-step-forward)))))

(defthm st=-step
  (implies (and (st= st1 st2)
                (syntaxp (not (term-order st2 st1))))
           (st= (system-step st2 input) (system-step st1 input)))
  :hints (("Goal" :expand ((st= (system-step st2 input)
                                (system-step st1 input))))))

; We don't need the following congruence rule as of this writing, but it's nice
; to be able to state our result as a congruence rule.  It's just an alternate
; form of the information-flow theorem just below.
(defcong st= st= (run st oracle) 1)

(defthm information-flow
  (implies (st= st1 st2)
          (st= (run st1 oracle) (run st2 oracle)))
  :rule-classes nil)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section 5: Non-interference theorem
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; We stub out the notion of a "high" node-id:

(defstub high (node-id) t)

; We now develop an invariant high-okp on the nodes of a state.  The idea is
; that any packet that has been marked (roughly, as "high security") is headed
; for a high-security destination.

(defund high-okp-pkt (pkt)
  (implies (marked pkt)
           (high (destination pkt))))

(defund high-okp-pkts (pkts)
  (if (endp pkts)
      t
    (and (high-okp-pkt (car pkts))
         (high-okp-pkts (cdr pkts)))))

(defund high-okp-body (node-id nodes)
  (and (equal (high node-id)
              (marked (g node-id nodes)))
       (high-okp-pkts (incoming-packets (g node-id nodes)))
       (high-okp-pkts (outgoing-packets (g node-id nodes)))))

(defun-sk high-okp (nodes)
  (forall node-id
          (high-okp-body node-id nodes)))

(in-theory (disable high-okp))

; The following predicate, low=, says that the two states satisfy our high-okp
; invariant and that the "low" view of each state, ignoring all marked data, is
; the same.

(defund low= (st1 st2)
  (and (high-okp (nodes st1))
       (high-okp (nodes st2))
       (st= st1 st2)))

; A secure oracle should never send information from a high source to a low
; destination.

(defun secure-oraclep (oracle)
  (if (endp oracle)
      t
    (and (implies (high (node-id (car oracle)))
                  (high (dest (car oracle))))
         (secure-oraclep (cdr oracle)))))

; Our goal is to prove the final theorem, non-interference, which says that our
; run function (interpreter) preserves the low= relation when given a secure
; oracle.

; Start proof of high-okp-system-step

; Start proof of high-okp-system-step-compute

(defthm outgoing-packets-set-local-state
  (equal (outgoing-packets (set-local-state ls node))
         (outgoing-packets node))
  :hints (("Goal" :in-theory (enable-theory accessors))))

(defthm incoming-packets-set-local-state
  (equal (incoming-packets (set-local-state ls node))
         (incoming-packets node))
  :hints (("Goal" :in-theory (enable-theory accessors))))

; Our proof of high-okp-system-step-compute, having been done first, does not
; quite following the pattern of the other three operations (below it).  When
; time permits it would be nice to redo this proof in the style of the other
; three.

(encapsulate
 ()

 (local
  (defthm high-okp-system-step-compute-lemma
    (implies (high-okp nodes)
             (high-okp-body wit
                            (s node-id
                               (set-local-state ls (g node-id nodes))
                               nodes)))
    :hints (("Goal" :use ((:theorem
                           (implies (high-okp nodes)
                                    (high-okp-body wit nodes))))
             :in-theory (enable high-okp-body)))))

 (defthm high-okp-system-step-compute
   (implies (high-okp (nodes st))
            (high-okp (nodes (system-step-compute st node-id))))
   :hints (("Goal"
            :in-theory (enable system-step-compute)
            :expand
            ((high-okp
              (s
              node-id
              (set-local-state (perform-computation
                                 (local-state (g node-id (nodes st))))
                                (g node-id (nodes st)))
              (nodes st))))))))

; Start proof of high-okp-system-step-create

(defthm incoming-packets-set-outgoing-packets
  (equal (incoming-packets (set-outgoing-packets pkts node))
         (incoming-packets node))
  :hints (("Goal" :in-theory (enable-theory accessors))))

(defthm outgoing-packets-set-outgoing-packets
  (equal (outgoing-packets (set-outgoing-packets pkts node))
         pkts)
  :hints (("Goal" :in-theory (enable-theory accessors))))

(defthm high-okp-pkts-enqueue
  (equal (high-okp-pkts (enqueue pkt pkts))
         (and (high-okp-pkt pkt)
              (high-okp-pkts pkts)))
  :hints (("Goal" :in-theory (enable high-okp-pkts))))

; Start proof of high-okp-pkt-new-packet

(defthm marked-set-marked
  (equal (marked (set-marked m node))
         m)
  :hints (("Goal" :in-theory (enable-theory accessors))))

(defthm destination-set-marked
  (equal (destination (set-marked m pkt))
         (destination pkt))
  :hints (("Goal" :in-theory (enable-theory accessors))))

(defthm destination-set-contents
  (equal (destination (set-contents c pkt))
         (destination pkt))
  :hints (("Goal" :in-theory (enable-theory accessors))))

(defthm destination-set-destination
  (equal (destination (set-destination dest pkt))
         dest)
  :hints (("Goal" :in-theory (enable-theory accessors))))

(defthm high-okp-body-s-node-id
  (equal (high-okp-body node-id1 (s node-id2 node nodes))
         (if (equal node-id1 node-id2)
             (and (equal (high node-id1)
                         (marked node))
```

```
                  (high-okp-pkts (incoming-packets node))
                  (high-okp-pkts (outgoing-packets node)))
           (high-okp-body node-id1 nodes)))
  :hints (("Goal" :in-theory (enable high-okp-body))))

(defthm high-okp-implies-marked-is-high
  (implies (high-okp nodes)
           (equal (marked (g node-id nodes))
                  (high node-id)))
  :hints (("Goal" :restrict ((high-okp-necc ((node-id node-id))))
           :in-theory (enable high-okp-body))))

(defthm high-okp-implies-high-okp-pkts-incoming-packets
  (implies (high-okp nodes)
           (high-okp-pkts (incoming-packets (g node-id nodes))))
  :hints (("Goal" :restrict ((high-okp-necc ((node-id node-id))))
           :in-theory (enable high-okp-body))))

(defthm high-okp-implies-high-okp-pkts-outgoing-packets
  (implies (high-okp nodes)
           (high-okp-pkts (outgoing-packets (g node-id nodes))))
  :hints (("Goal" :restrict ((high-okp-necc ((node-id node-id))))
           :in-theory (enable high-okp-body))))

; Start proof of high-okp-system-step-create-lemma

(defthm destination-new-packet
  (equal (destination (new-packet node-id dest next-pkt-id nodes))
         dest)
  :hints (("Goal" :in-theory (enable new-packet))))

(defthm marked-new-packet
  (equal (marked (new-packet node-id dest next-pkt-id nodes))
         (marked (g node-id nodes)))
  :hints (("Goal" :in-theory (enable new-packet))))

(defthm high-okp-pkt-new-packet
  (implies (and (high-okp (nodes st))
                (not (and (high node-id)
                          (not (high dest)))))
           (high-okp-pkt (new-packet node-id dest (next-packet-id st)
                                     (nodes st))))
  :hints (("Goal" :in-theory (enable high-okp-pkt))))

(encapsulate
 ()

 (local
  (defthm high-okp-system-step-create-lemma
    (implies
     (and (high-okp (nodes st))
          (not (and (high node-id)
                    (not (high dest)))))
     (high-okp-body
      (high-okp-witness (nodes (system-step-create st node-id dest)))
      (nodes (system-step-create st node-id dest))))
    :hints (("Goal" :in-theory (enable system-step-create)))))

 (defthm high-okp-system-step-create
   (implies (and (high-okp (nodes st))
                 (not (and (high node-id)
                           (not (high dest)))))
            (high-okp (nodes (system-step-create st node-id dest))))
   :hints (("Goal"
            :expand
            ((high-okp (nodes (system-step-create st node-id dest))))))))

; Start proof of high-okp-system-step-consume

(defthm incoming-packets-node-consume-packet
  (equal (incoming-packets (node-consume-packet pkt node))
         (incoming-packets node))
  :hints (("Goal" :in-theory (enable-theory accessors (node-consume-packet)))))

(defthm incoming-packets-set-incoming-packets
  (equal (incoming-packets (set-incoming-packets pkts node))
         pkts)
  :hints (("Goal" :in-theory (enable-theory accessors))))

(defthm marked-node-consume-packet
  (equal (marked (node-consume-packet pkt node))
         (or (marked node) (marked pkt)))
  :hints (("Goal" :in-theory (enable-theory accessors (node-consume-packet)))))

(defthm outgoing-packets-node-consume-packet
  (equal (outgoing-packets (node-consume-packet pkt node))
         (outgoing-packets node))
  :hints (("Goal" :in-theory (enable-theory accessors (node-consume-packet)))))

(defthm high-okp-pkts-cdr
  (implies (high-okp-pkts pkts)
           (high-okp-pkts (cdr pkts)))
  :hints (("Goal" :in-theory (enable high-okp-pkts))))

(defthm outgoing-packets-set-incoming-packets
  (equal (outgoing-packets (set-incoming-packets pkts node))
         (outgoing-packets node))
  :hints (("Goal" :in-theory (enable-theory accessors))))

(defthm marked-set-incoming-packets
  (equal (marked (set-incoming-packets pkts node))
         (marked node))
  :hints (("Goal" :in-theory (enable-theory accessors))))

(defthm high-okp-implies-high-okp-pkt-car-incoming-packets
  (implies (and (high-okp nodes)
                (consp (incoming-packets (g node-id nodes))))
           (high-okp-pkt (car (incoming-packets (g node-id nodes)))))
  :hints (("Goal" :use high-okp-implies-high-okp-pkts-incoming-packets
           :in-theory
           (e/d (high-okp-pkts)
                (high-okp-implies-high-okp-pkts-incoming-packets)))))

(defthm high-okp-implies-car-incoming-packets-not-marked
  (let* ((pkts (incoming-packets (g node-id nodes)))
         (pkt (car pkts)))
    (implies (and (high-okp nodes)
                  (consp pkts)
                  (not (high (destination pkt))))
             (not (marked pkt))))
  :hints (("Goal" :use high-okp-implies-high-okp-pkt-car-incoming-packets
           :in-theory (enable high-okp-pkt))))

(encapsulate
 ()

 (local
  (defthm high-okp-system-step-consume-lemma
    (implies
     (high-okp (nodes st))
     (high-okp-body (high-okp-witness (nodes (system-step-consume st node-id)))
                    (nodes (system-step-consume st node-id))))
    :hints (("Goal" :in-theory (enable system-step-consume)))))

 (defthm high-okp-system-step-consume
   (implies (high-okp (nodes st))
            (high-okp (nodes (system-step-consume st node-id))))
   :hints (("Goal"
            :expand
            ((high-okp (nodes (system-step-consume st node-id))))))))

; Start proof of high-okp-system-step-forward

(defthm high-okp-implies-high-okp-pkt-car-outgoing-packets
  (implies (and (high-okp nodes)
                (consp (outgoing-packets (g node-id nodes))))
           (high-okp-pkt (car (outgoing-packets (g node-id nodes)))))
  :hints (("Goal" :use high-okp-implies-high-okp-pkts-outgoing-packets
           :in-theory
           (e/d (high-okp-pkts)
                (high-okp-implies-high-okp-pkts-outgoing-packets)))))

(encapsulate
 ()

 (local
  (defthm high-okp-system-step-forward-lemma
    (implies (high-okp (nodes st))
             (high-okp-body (high-okp-witness
                              (nodes (system-step-forward st node-id)))
                             (nodes (system-step-forward st node-id))))
    :hints (("Goal" :in-theory (enable system-step-forward)))))

 (defthm high-okp-system-step-forward
   (implies (high-okp (nodes st))
            (high-okp (nodes (system-step-forward st node-id))))
   :hints (("Goal"
            :expand ((high-okp (nodes (system-step-forward st node-id)))))))

(defthm high-okp-system-step
   (implies (and (high-okp (nodes st))
                 (not (and (high (node-id input))
                           (not (high (dest input))))))
            (high-okp (nodes (system-step st input))))
   :hints (("Goal" :in-theory (enable system-step))))

(defthm high-okp-run
  (implies (and (high-okp (nodes st))
                (secure-oraclep oracle))
           (high-okp (nodes (run st oracle)))))

(defthm non-interference
  (implies (and (low= st1 st2)
                (secure-oraclep oracle))
           (low= (run st1 oracle)
                 (run st2 oracle)))
  :hints (("Goal" :in-theory (enable low=))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section 6: Corollaries for initial states
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Finally, we introduce the notion of an initial state, and use it to state
; the desired corollaries of our main theorems.

; We use the prefix "r-" for "restricted", meaning that we don't care about the
; packet queues.  We consider two nodes to be r-node= if they agree on their
; routing tables and on whether they are marked, and if not marked then they
; also agree on their local states -- but their queues are irrelevant for
; r-node=.  Then we correspondingly restrict nodes=, st=, and low= to notions
; r-nodes=, r-st=, and r-low=, respectively.

; Our goals are then the following two corollaries of the corresponding
; theorems about (with "r-" deleted):
```

```
; Theorem r-information-flow:
; Suppose we run our interpreter on marked-equivalent (r-st=) states st1 and
; st2 with the same inputs (oracle).  Then the two final states returned are
; marked-equivalent (r-st=).

; Theorem r-non-interference:
; Suppose we run our interpreter on low-equivalent (r-low=) states st1 and st2
; with the same input sequence (oracle), where no element of that sequence
; creates a packet from a high source to a low destination.  Then the two final
; states returned are low-equivalent (r-low=).

(defun-sk initial-statep (st)
  (forall node-id
          (let ((node (g node-id (nodes st))))
            (and (null (incoming-packets node))
                 (null (outgoing-packets node))))))

(defund r-erase-incoming-queues (node)
  (set-incoming-packets nil node))

(defund r-erase-outgoing-queues (node)
  (set-outgoing-packets nil node))

(defund r-erase-node (node)
  (erase-local-state
   (r-erase-incoming-queues
    (r-erase-outgoing-queues
     node))))

(defund r-node= (node1 node2)
  (equal (r-erase-node node1)
         (r-erase-node node2)))

(defun-sk r-nodes= (nodes1 nodes2)
  (forall node-id
          (r-node= (g node-id nodes1) (g node-id nodes2)))
  :rewrite :direct)

(defun r-st= (st1 st2)
  (and (equal (next-packet-id st1) (next-packet-id st2))
       (r-nodes= (nodes st1) (nodes st2))))

(defthm node=-implies-r-node=-helper
  (equal (erase-local-state
          (r-erase-incoming-queues
           (r-erase-outgoing-queues n)))
         (r-erase-incoming-queues
          (r-erase-outgoing-queues
           (erase-local-state
            (erase-incoming-queues
            (erase-outgoing-queues n))))))
  :hints (("Goal" :in-theory (enable marked
                                     erase-local-state
                                     erase-incoming-queues
                                     erase-outgoing-queues
                                     r-erase-incoming-queues
                                     r-erase-outgoing-queues
                                     set-local-state
                                     set-incoming-packets
                                     set-outgoing-packets))))


(defthm node=-implies-r-node=
  (implies (node= n1 n2)
           (r-node= n1 n2))
  :hints (("Goal" :in-theory (enable node=
                                     r-node=
                                     erase-node
                                     r-erase-node))))

(defthm st=-implies-r-st=-lemma
  (implies (nodes= nodes1 nodes2)
           (r-node= (g node-id nodes1)
                    (g node-id nodes2)))
  :hints (("Goal" :use nodes=-necc
           :in-theory (disable nodes=-implies-node=-g-2
                               nodes=-forward))))

(defthm st=-implies-r-st=
  (implies (st= st1 st2)
           (r-st= st1 st2))
  :hints (("Goal" :in-theory (enable st=))))

(in-theory (disable initial-statep r-nodes=))

(defthm erase-incoming-queues-noop
  (implies (not (incoming-packets node))
           (equal (erase-incoming-queues node)
                  node))
  :hints (("Goal" :in-theory (e/d (incoming-packets
                                  erase-incoming-queues
                                  set-incoming-packets)
                                 (s-same-g))
           :use ((:instance s-same-g
                            (r node)
                            (a :incoming-packets))))))

(defthm erase-outgoing-queues-noop
  (implies (not (outgoing-packets node))
           (equal (erase-outgoing-queues node)
                  node))
  :hints (("Goal" :in-theory (e/d (outgoing-packets
                                   erase-outgoing-queues
                                   set-outgoing-packets)
                                  (s-same-g))
           :use ((:instance s-same-g
                            (r node)
                            (a :outgoing-packets))))))

(defthm r-erase-incoming-queues-noop
  (implies (not (incoming-packets node))
           (equal (r-erase-incoming-queues node)
                  node))
  :hints (("Goal" :in-theory (e/d (incoming-packets
                                   r-erase-incoming-queues
                                   set-incoming-packets)
                                  (s-same-g))
           :use ((:instance s-same-g
                            (r node)
                            (a :incoming-packets))))))

(defthm r-erase-outgoing-queues-noop
  (implies (not (outgoing-packets node))
           (equal (r-erase-outgoing-queues node)
                  node))
  :hints (("Goal" :in-theory (e/d (outgoing-packets
                                   r-erase-outgoing-queues
                                   set-outgoing-packets)
                                  (s-same-g))
           :use ((:instance s-same-g
                            (r node)
                            (a :outgoing-packets))))))

(defthm r-erase-outgoing-queues-erase-local-state
  (equal (r-erase-outgoing-queues (erase-local-state node))
         (erase-local-state (r-erase-outgoing-queues node)))
  :hints (("Goal" :in-theory (enable erase-local-state
                                     r-erase-outgoing-queues
                                     set-local-state
                                     set-outgoing-packets
                                     marked))))

(defthm r-erase-incoming-queues-erase-local-state
  (equal (r-erase-incoming-queues (erase-local-state node))
         (erase-local-state (r-erase-incoming-queues node)))
  :hints (("Goal" :in-theory (enable erase-local-state
                                     r-erase-incoming-queues
                                     set-local-state
                                     set-incoming-packets
                                     marked))))

(defthm r-node=-implies-node=-sometimes
  (implies (and (r-node= (g node-id (nodes st1))
                         (g node-id (nodes st2)))
                (not (incoming-packets (g node-id (nodes st1))))
                (not (outgoing-packets (g node-id (nodes st1))))
                (not (incoming-packets (g node-id (nodes st2))))
                (not (outgoing-packets (g node-id (nodes st2)))))
           (equal (node= (g node-id (nodes st1))
                         (g node-id (nodes st2)))
                  t))
  :hints (("Goal" :in-theory (e/d (node=
                                   r-node=
                                   erase-node
                                   r-erase-node)
                                  (node=-implies-r-node=-helper)))))

(encapsulate
 ()

; Start proof of initial-statep-implies-r-st=-implies-st=.

 (local
  (defthm lemma-1
    (implies (and (initial-statep st1)
                  (initial-statep st2)
                  (r-nodes= (nodes st1) (nodes st2)))
             (equal (node= (g node-id (nodes st1))
                           (g node-id (nodes st2)))
                    t))
    :hints (("Goal" :in-theory (disable r-nodes=-necc initial-statep-necc)
             :use ((:instance r-nodes=-necc
                              (nodes1 (nodes st1))
                              (nodes2 (nodes st2)))
                   (:instance initial-statep-necc (st st1))
                   (:instance initial-statep-necc (st st2)))))))

 (local
  (defthm initial-statep-implies-r-st=-implies-st=
    (implies (and (initial-statep st1)
                  (initial-statep st2))
             (implies (r-st= st1 st2)
                      (equal (st= st1 st2)
                             t)))
    :hints (("Goal" :in-theory (enable st= nodes=)))))

 (defthm initial-statep-implies-st=-iff-r-st=
   (implies (and (initial-statep st1)
                 (initial-statep st2))
            (iff (st= st1 st2)
                 (r-st= st1 st2))))

; Finally, here is our first main corollary.
```

```
(defthm r-information-flow
  (implies (and (initial-statep st1)
                (initial-statep st2)
                (r-st= st1 st2))
           (r-st= (run st1 oracle) (run st2 oracle)))
  :hints (("Goal" :use information-flow))
  :rule-classes nil)

; Now on to noninterference.

(defun-sk r-high-okp (nodes)
  (forall node-id
          (equal (high node-id)
                 (marked (g node-id nodes)))))

(in-theory (disable r-high-okp))

(defund r-low= (st1 st2)
  (and (r-high-okp (nodes st1))
       (r-high-okp (nodes st2))
       (r-st= st1 st2)))

(defthm low=-implies-r-low=
  (implies (low= st1 st2)
           (r-low= st1 st2))
  :hints (("Goal" :in-theory (enable low= r-low= r-high-okp))))

(defthm high-okp=-implies-r-high-okp=
  (implies (high-okp (nodes st))
           (r-high-okp (nodes st)))
  :hints (("Goal" :in-theory (enable r-high-okp))))

(defthm initial-statep-implies-null-incoming-packets
  (implies (initial-statep st)
           (equal (incoming-packets (g node-id (nodes st)))
                  nil))
  :hints (("Goal" :restrict ((initial-statep-necc
                              ((node-id node-id)))))))

(defthm initial-statep-implies-null-outgoing-packets
  (implies (initial-statep st)
           (equal (outgoing-packets (g node-id (nodes st)))
                  nil))
  :hints (("Goal" :restrict ((initial-statep-necc
                              ((node-id node-id)))))))

(defthm initial-statep-implies-r-high-okp=-implies-high-okp=
  (implies (initial-statep st)
           (implies (r-high-okp (nodes st))
                    (high-okp (nodes st))))
  :hints (("Goal" :in-theory (e/d (high-okp-body
                                    high-okp)
                                   (r-high-okp-necc))
           :use ((:instance r-high-okp-necc
                            (node-id (high-okp-witness (nodes st)))
                            (nodes (nodes st)))))))

(defthm initial-statep-implies-low=-iff-r-low=
  (implies (and (initial-statep st1)
                (initial-statep st2))
           (iff (low= st1 st2)
                (r-low= st1 st2)))
  :hints (("Goal" :in-theory (enable low= r-low=))))

(defthm r-non-interference
  (implies (and (r-low= st1 st2)
                (secure-oraclep oracle)
                (initial-statep st1)
                (initial-statep st2))
           (r-low= (run st1 oracle)
                   (run st2 oracle)))
  :hints (("Goal" :in-theory (enable low=))))
```